---

This book is available in English.

Full translation available in Deutsch, 简体中文, 正體中文, Français, 日本語, Nederlands, Русский, 한국어, Português (Brasil) and Čeština.

Partial translations available in Arabic, Español, Indonesian, Italiano, Suomi, Македонски, Ελληνικά, Polski and Türkçe.

Translations started for Azərbaycan dili, Беларуская, Català, Esperanto, Español (Nicaragua), فارسی, हिन्दी, Magyar, Norwegian Bokmål, Română, Српски, ภาษาไทย, Tiếng Việt, Українська and Ўзбекча.

---

The source of this book is hosted on GitHub.
Patches, suggestions and comments are welcome.

Chapters ▾

1st Edition

# 3.2 Git Branching - Einfaches Branching und Merging

## Einfaches Branching und Merging

Lass uns das Ganze an einem Beispiel durchgehen, dessen Workflow zum Thema Branching und Zusammenführen Du im echten Leben verwenden kannst. Folge einfach diesen Schritten:

1. Arbeite an einer Webseite.
2. Erstell einen Branch für irgendeine neue Geschichte, an der Du arbeitest.
3. Arbeite in dem Branch.

In diesem Augenblick kommt ein Anruf, dass ein kritisches Problem aufgetreten ist und sofort gelöst werden muss. Du machst folgendes:

1. Schalte zurück zu Deinem „Produktiv“-Zweig.
2. Erstelle eine Branch für den Hotfix.
3. Nach dem Testen führst Du den Hotfix-Branch mit dem „Produktiv“-Branch zusammen.
4. Schalte wieder auf Deine alte Arbeit zurück und werkel weiter.

## Branching Grundlagen

Sagen wir, Du arbeitest an Deinem Projekt und hast bereits einige Commits durchgeführt (siehe Abbildung 3-10).

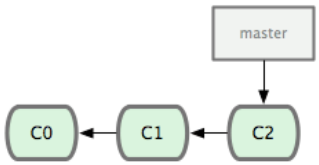


Abbildung 3-10. Eine kurze, einfache Commit-Historie

Du hast Dich dafür entschieden an dem Issue #53, des Issue-Trackers XY, zu arbeiten. Um eines klarzustellen, Git ist an kein Issue-Tracking-System gebunden. Da der Issue #53 allerdings ein Schwerpunktthema betrifft, wirst Du einen neuen Branch erstellen um daran zu arbeiten. Um in einem Arbeitsschritt einen neuen Branch zu erstellen und zu aktivieren kannst Du das Kommando `git checkout` mit der Option `-b` verwenden:

```
$ git checkout -b iss53
Switched to a new branch 'iss53'
```

Das ist die Kurzform von

```
$ git branch iss53
$ git checkout iss53
```

Abbildung 3-11 verdeutlicht das Ergebnis.

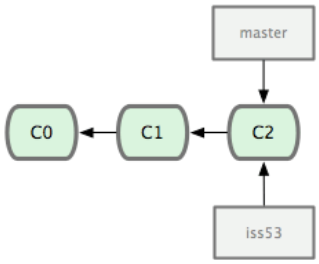


Abbildung 3-11. Erstellung eines neuen Branch-Zeigers

Du arbeitest an Deiner Web-Seite und machst ein paar Commits. Das bewegt den `iss53`-Branch vorwärts, da Du ihn ausgebucht hast (das heißt, dass Dein HEAD-Zeiger darauf verweist; siehe Abbildung 3-12):

```
$ vim index.html
$ git commit -a -m 'added a new footer [issue 53]'
```

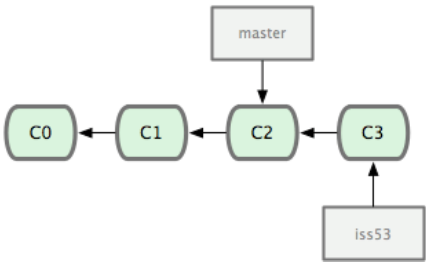


Abbildung 3-12. Der `iss53`-Branch hat mit Deiner Arbeit Schritt gehalten.

Nun bekommst Du einen Anruf, in dem Dir mitgeteilt wird, dass es ein Problem mit der Internet-Seite gibt, das Du umgehend beheben sollst. Mit Git musst Du Deine Fehlerkorrektur nicht zusammen mit den `iss53`-Änderungen einbringen. Und Du musst keine Zeit damit verschwenden Deine bisherigen Änderungen rückgängig zu machen, bevor Du mit der Fehlerbehebung an der Produktionsumgebung beginnen kannst. Alles was Du tun musst, ist zu Deinem MASTER-Branch wechseln.

Beachte jedoch, dass Dich Git den Branch nur wechseln lässt, wenn bisherige Änderungen in Deinem Arbeitsverzeichnis oder Deiner Staging-Area nicht in Konflikt mit dem Zweig stehen, zu dem Du nun wechseln

möchtest. Am besten es liegt ein sauberer Status vor wenn man den Branch wechselt. Wir werden uns später mit Wegen befassen, dieses Verhalten zu umgehen (namentlich „Stashing“ und „Commit Ammending“). Vorerst aber hast Du Deine Änderungen bereits comitted, sodass Du zu Deinem MASTER-Branch zurückwechseln kannst.

```
$ git checkout master
Switched to branch 'master'
```

Zu diesem Zeitpunkt befindet sich das Arbeitsverzeichnis des Projektes in exakt dem gleichen Zustand, in dem es sich befand, als Du mit der Arbeit an Issue #53 begonnen hast und Du kannst Dich direkt auf Deinen Hotfix konzentrieren. Dies ist ein wichtiger Moment um sich vor Augen zu halten, dass Git Dein Arbeitsverzeichnis auf den Zustand des Commits, auf den dieser Branch zeigt, zurücksetzt. Es erstellt, entfernt und verändert Dateien automatisch, um sicherzustellen, dass Deine Arbeitskopie haargenau so aussieht wie der Zweig nach Deinem letzten Commit.

Nun hast Du einen Hotfix zu erstellen. Lass uns dazu einen Hotfix-Branch erstellen, an dem Du bis zu dessen Fertigstellung arbeitest (siehe Abbildung 3-13):

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix 3a0874c] fixed the broken email address
 1 files changed, 1 deletion(-)
```

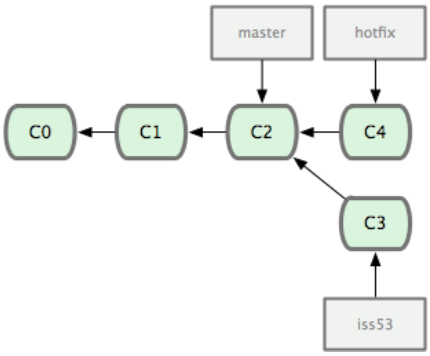


Abbildung 3-13. Der Hotfix-Branch basiert auf dem zurückliegenden Master-Branch.

Mach Deine Tests, stell sicher das sich der Hotfix verhält wie erwartet und führe ihn mit dem Master-Branch zusammen, um ihn in die Produktionsumgebung zu integrieren. Das machst Du mit dem `git merge`-Kommando:

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 README | 1 -
 1 file changed, 1 deletion(-)
```

Du wirst die Mitteilung „Fast Forward“ während des Zusammenführens bemerken. Da der neue Commit direkt von dem ursprünglichen Commit, auf den sich der nun eingebrachte Zweig bezieht, abstammt, bewegt Git einfach den Zeiger weiter. Mit anderen Worten kann Git den neuen Commit, durch Verfolgen der Commitabfolge, direkt erreichen, dann bewegt es ausschließlich den Branch-Zeiger. Zu einer tatsächlichen Kombination der Commits besteht ja kein Anlass. Dieses Vorgehen wird „Fast Forward“ genannt.

Deine Modifikationen befinden sich nun als Schnappschuss in dem Commit, auf den der `master`-Branch zeigt, diese lassen sich nun veröffentlichen (siehe Abbildung 3-14).

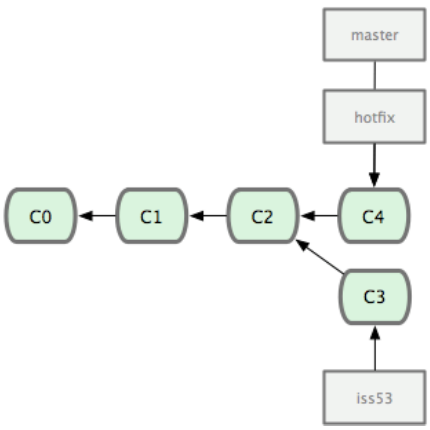


Abbildung 3-14. Der Master-Branch zeigt nach der Zusammenführung auf den gleichen Commit wie der Hotfix-

Branch.

Nachdem Dein superwichtiger Hotfix veröffentlicht wurde, kannst Du Dich wieder Deiner ursprünglichen Arbeit zuwenden. Vorher wird sich allerdings des nun nutzlosen Hotfix-Zweiges entledigt, schließlich zeigt der Master-Branch ebenfalls auf die aktuelle Version. Du kannst ihn mit der `-d`-Option von `git branch` entfernen:

```
$ git branch -d hotfix
Deleted branch hotfix (was 3a0874c).
```

Nun kannst Du zu Deinem Issue #53-Branch zurückwechseln und mit Deiner Arbeit fortfahren (Abbildung 3-15):

```
$ git checkout iss53
Switched to branch 'iss53'
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53 ad82d7a] finished the new footer [issue 53]
 1 file changed, 1 insertion(+)
```

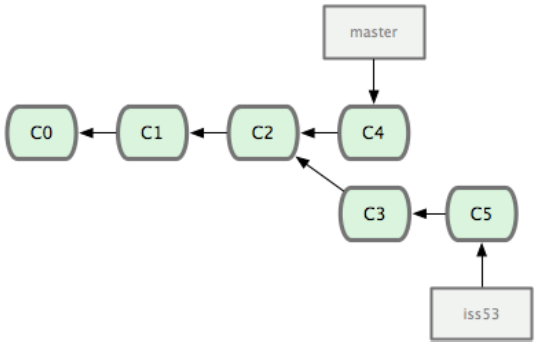


Abbildung 3-15. Dein `iss53`-Branch kann sich unabhängig weiterentwickeln.

An dieser Stelle ist anzumerken, dass die Änderungen an dem `hotfix`-Branch nicht in Deinen `iss53`-Zweig eingeflossen sind. Falls nötig kannst Du den `master`-Branch allerdings mit dem Kommando `git merge master` mit Deinem Zweig kombinieren. Oder Du wartest, bis Du den `iss53`-Branch später in den Master-Zweig zurückführst.

### Die Grundlagen des Zusammenführens (Mergen)

Angenommen Du entscheidest dich, dass Deine Arbeit an issue #53 getan ist und Du diese mit der `master` Branch zusammenführen möchtest. Das passiert, indem Du ein `merge` in die `iss53` Branch machst, ähnlich dem `merge` mit der `hotfix` Branch von vorhin. Alles was Du machen musst, ist ein `checkout` der Branch, in die Du das `merge` machen willst und das Ausführen des Kommandos `git merge`:

```
$ git checkout master
$ git merge iss53
Auto-merging README
Merge made by the 'recursive' strategy.
 README | 1 +
 1 file changed, 1 insertion(+)
```

Das sieht ein bisschen anders aus als das `hotfix merge` von vorhin. Hier läuft Deine Entwicklungshistorie auseinander. Ein `commit` auf Deine Arbeits-Branch ist kein direkter Nachfolger der Branch in die Du das `merge` gemacht hast, Git hat da einiges zu tun, es macht einen 3-Wege `merge`: es geht von den beiden `snapshots` der Branches und dem allgemeinen Nachfolger der beiden aus. Abbildung 3-16 zeigt die drei `snapshots`, die Git in diesem Fall für das `merge` verwendet.

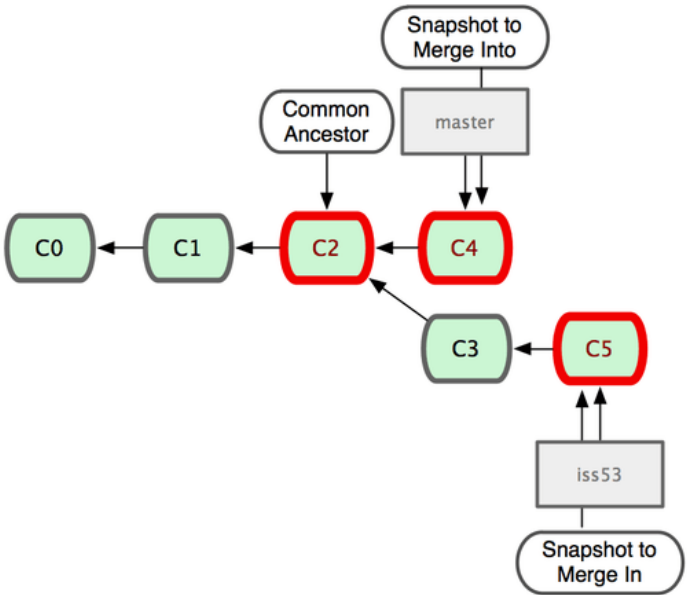


Abbildung 3-16. Git ermittelt automatisch die beste Nachfolgebasis für die Branchzusammenführung.

Anstatt einfach den 'pointer' weiterzubewegen, erstellt Git einen neuen `snapshot`, der aus dem 3-Wege 'merge' resultiert und erzeugt einen neuen 'commit', der darauf verweist (siehe Abbildung 3-17). Dies wird auch als 'merge commit' bezeichnet und ist ein Spezialfall, weil es mehr als nur ein Elternteil hat.

Es ist wichtig herauszustellen, dass Git den besten Nachfolger für die 'merge' Basis ermittelt, denn hierin unterscheidet es sich von CVS und Subversion (vor Version 1.5), wo der Entwickler die 'merge' Basis selbst ermitteln muss. Damit wird das Zusammenführen in Git um einiges leichter, als in anderen Systemen.

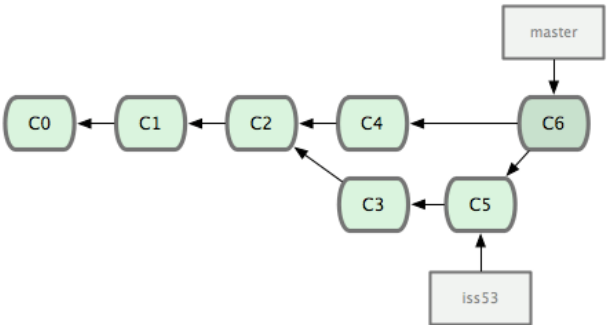


Abbildung 3-17. Git erstellt automatisch ein 'commit', dass die zusammengeführte Arbeit enthält.

Jetzt da wir die Arbeit zusammengeführt haben, ist der `iss53`-Branch nicht mehr notwendig. Du kannst ihn löschen und das Ticket im Ticket-Tracking-System schliessen.

```
$ git branch -d iss53
```

## Grundlegende Merge-Konflikte

Gelegentlich verläuft der Prozess nicht ganz so glatt. Wenn Du an den selben Stellen in den selben Dateien unterschiedlicher Branches etwas geändert hast, kann Git diese nicht sauber zusammenführen. Wenn Dein Fix an 'issue #53' die selbe Stelle in einer Datei verändert hat, die Du auch mit `hotfix` angefasst hast, wirst Du einen 'merge'-Konflikt erhalten, der ungefähr so aussehen könnte:

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Git hat hier keinen 'merge commit' erstellt. Es hat den Prozess gestoppt, damit Du den Konflikt beseitigen kannst. Wenn Du sehen willst, welche Dateien 'unmerged' aufgrund eines 'merge' Konflikts sind, benutze einfach `git status`:

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
```

```
  (use "git add <file>..." to mark resolution)

        both modified:      index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

Alles, was einen 'merge' Konflikt aufweist und nicht gelöst werden konnte, wird als 'unmerged' aufgeführt. Git fügt den betroffenen Dateien Standard-Konfliktlösungsmarker hinzu, sodass Du diese öffnen und den Konflikt manuell lösen kannst. Deine Datei enthält einen Bereich, der so aussehen könnte:

```
<<<<<<< HEAD
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
  please contact us at support@github.com
</div>
>>>>>>> iss53
```

Das heisst, die Version in HEAD (Deines 'master'-Branches, denn der wurde per 'checkout' aktiviert als Du das 'merge' gemacht hast) ist der obere Teil des Blocks (alles oberhalb von '========'), und die Version aus dem `iss53`-Branch sieht wie der darunter befindliche Teil aus. Um den Konflikt zu lösen, musst Du Dich entweder für einen der beiden Teile entscheiden oder Du ersetzt den Teil komplett:

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

Diese Lösung hat von beiden Teilen etwas und ich habe die Zeilen mit `<<<<<<<`, `=======`, und `>>>>>>>` komplett gelöscht. Nachdem Du alle problematischen Bereiche, in allen durch den Konflikt betroffenen Dateien, beseitigt hast, führe einfach `git add` für alle betroffenen Dateien aus und markieren sie damit als bereinigt. Dieses 'staging' der Dateien markiert sie für Git als bereinigt. Wenn Du ein grafischen Tool zur Bereinigung benutzen willst, dann verwende `git mergetool`. Das welches ein passendes grafisches 'merge'-Tool startet und Dich durch die Konfliktbereiche führt:

```
$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge p4merge araxis bc3 codecompare vimdiff emerge
Merging:
index.html

Normal merge conflict for 'index.html':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (opendiff):
```

Wenn Du ein anderes Tool anstelle des Standardwerkzeug für ein 'merge' verwenden möchtest (Git verwendet in meinem Fall `opendiff`, da ich auf einem Mac arbeite), dann kannst Du alle unterstützten Werkzeuge oben – unter „one of the following tools“ – aufgelistet sehen. Tippe einfach den Namen Deines gewünschten Werkzeugs ein. In Kapitel 7 besprechen wir, wie Du diesen Standardwert in Deiner Umgebung dauerhaft ändern kannst.

Wenn Du das 'merge' Werkzeug beendest, fragt Dich Git, ob das Zusammenführen erfolgreich war. Wenn Du mit 'Ja' antwortest, wird das Skript diese Dateien als gelöst markieren.

Du kannst `git status` erneut ausführen, um zu sehen, ob alle Konflikte gelöst sind:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   index.html
```

Wenn Du zufrieden bist und Du geprüft hast, dass alle Konflikte beseitigt wurden, kannst Du `git commit` ausführen um den 'merge commit' abzuschliessen. Die Standardbeschreibung für diese Art 'commit' sieht wie folgt aus:

```
Merge branch 'iss53'

Conflicts:
  index.html
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
#       .git/MERGE_HEAD
# and try again.
#
```

Wenn Du glaubst für zukünftige Betrachter des Commits könnte interessant sein warum Du getan hast, was Du getan hast, dann kannst Du der Commit-Beschreibung noch zusätzliche Informationen hinzufügen – sofern das nicht trivial erscheint.

prev | next

Patches, suggestions and comments are welcome.
Git is a member of Software Freedom Conservancy